NBSIR 88-3776

The GRAMPS Operating System: User's Guide

Peter Mansbach

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Manufacturing Engineering
Robot Systems Division
Gaithersburg, MD 20899

and

Michael Shneier

Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510

September 1988

75 Years Stimulating America's Progress
1913-1988

U.S. DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS

NBSIR 88-3776

THE GRAMPS OPERATING SYSTEM: USER'S GUIDE

Peter Mansbach

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Manufacturing Engineering
Robot Systems Division
Gaithersburg, MD 20899

and

Michael Shneier

Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510

September 1988

U.S. DEPARTMENT OF COMMERCE, C. William Verity, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

The GRAMPS Operating System: User's Guide

Peter Mansbach[†] *and Michael Shneier*[*]

[†] Sensory-Interactive Robotics Group
National Bureau of Standards
Bldg. 220/Rm. B-124
Gaithersburg, MD 20899

[*] Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510

ABSTRACT

This guide describes the *GRAMPS* real-time multiprocessor operating system from an applications viewpoint. It presents the information needed to use *GRAMPS* in implementing distributed processing applications. Additional information needed by an administrator to set up and maintain a specific application appears in the *Administrator's Guide*.

April 15, 1988

The GRAMPS Operating System: User's Guide

Peter Mansbach[†] *and Michael Shneier*[*]

[†] Sensory-Interactive Robotics Group
National Bureau of Standards
Bldg. 220/Rm. B-124
Gaithersburg, MD 20899

[*] Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510

System Overview

GRAMPS (for General Real-time Asynchronous Multi-Processor System) is a distributed operating system developed at the National Bureau of Standards *[Ref. 1]* and designed to allow applications involving multiple processors to be implemented easily. A typical system using *GRAMPS* will consist of several processors (*Motorola 680x0* in the current implementation[1]), a backplane (*VMEbus*), and a number of memory boards, special-purpose peripherals, and I/O ports. *GRAMPS* provides facilities for passing information from one process to another, for sharing dynamically allocated memory among processes, and for communicating with users and peripherals.

GRAMPS does not provide a development environment, although many debugging facilities have been implemented. The applications must be programmed on a host computer (e.g., a Sun) and be compiled, linked with the *GRAMPS* library, and downloaded (preferably using *GRAMPS*' downloader) onto the target boards. Once downloaded, the programs will run stand-alone, and communicate with each other and with the user.

GRAMPS is a true distributed operating system, in that each processor contains a part of the operating system. Each processor can run by itself, in the absence of other processors, without risk of failure, and processors can be downloaded and started in any order, with the same system characteristics guaranteed to exist when the total system is running. In addition, each processor will contain code only for those sections of the operating system that it actually uses, ensuring the least possible impact of the operating system on the size of the code.

GRAMPS is deliberately not a multi-tasking system. It was felt that the overhead involved in task switching was an undesirable burden. Also, the code for I/O operations becomes much more complex in a multi-tasking system, particularly when a communication my arrive while the task expecting it is not running, or worse yet, swapped out. Keeping these operations simple results in faster execution, smaller code, and easier modification by users.

In response to user requests, provision for simple task-switching, under user control, has been added. A timer-initiated task switch is also provided. It is thus possible for the user to create a customized multi-tasking system.

Two protocols are provided for processes to communicate. The primary (and only necessary) way is through "files". A file is an area of common memory, accessible to all the processes that will use it, and known to them by a name, which is an ASCII string. A file is implemented as a common

[1] Commercial products are identified in order adequately to describe the equipment. In no case does such identification imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the equipment identified is necessarily the best available for the purpose.

memory buffer, an area of fixed size and location. Its use, however, is more like a file on an ordinary operating system, such as *Unix*[2], except that it only exists in (volatile) memory. A process can open a file, write or read the information in it either randomly or sequentially, and close the file. The system ensures that no two processes can write the same file at the same time (or one read while the other writes).

Information is passed from one process to another by writing it into the file, which must be read by the other process. (Note that there is no requirement that the processes run on different processors.) The system does not insist that something be written in the file before the reader can access it, so the user must make sure that there is useful information available before reading. This is usually done by checking flags. The identity of the previous user of a file is always available (in the flag *previous_user*), and is updated every time the file is opened. Thus, after writing data into a file, the identifier of the writing process is associated with the file until some other process opens the file, either to read or to write. It is left to the implementor to decide whether or not processes should be allowed to overwrite information before it has been read, or to read information that was seen before. In appropriate instances, both preventing and allowing these operations may be useful, and are easily accommodated.

The second way that processes may interact is through the use of dynamically allocated structures in memory. *GRAMPS* includes two kinds of dynamic memory allocation. The first is a purely local system, internal to a process, in which memory may be allocated in a private area, and is not accessible by other processes. The second is a common memory allocation system, in which a process can allocate memory, pass a pointer to the allocated area to other processes, and, eventually, free the memory. Each process that receives a pointer to the information must free the area before the memory is returned to the free list, thus guaranteeing that structures will be available as long as any process has need of them. Allocation and freeing are managed in a way similar to that used for links to files in *Unix*. A bit is set (normally by the allocating process) for each process which will use an allocated area. As each process frees the area, its bit is turned off. When all bits are off, the area is finally returned to the free list.

It is obviously necessary to pass the address of the dynamically allocated area to each of the processes that will use it. This is done by means of a file, so that files are the more basic means of data transfer. Once a process has read the address, it may access the structure, modify it, and pass it on to other processes. There is not as much protection for these structures as there is for files, in that there is no attempt to prevent multiple writers or readers from simultaneously accessing the same structure. If this is a problem, a dummy file can be set up, which must be opened or closed before accessing the structure. This provides the same protection for a dynamically allocated structure as is given to normal files. Note that this is considered an implementation-level decision, rather than being part of the operating system.

In addition to communicating with other processes, access is provided to whatever serial ports may be available to the processor. These ports may be written and read as if they were files. Standard input and output files are provided by the system, and usually bound to the terminal port. When special peripherals are available, it is necessary to write simple drivers that either use fixed memory locations for communication, or that have their own processors to interact with the system.

The rest of this document provides a guide to using the system for the general user. It explains how to create and run a process, and describes tools which may be used in debugging it. Examples of simple programs are given to illustrate some of the concepts. Appendix A gives a list of the *GRAMPS* functions of interest to the user, together with brief descriptions of what they do. Appendix B is a listing of the header file **vbus.h**, which must be **#include**'d in user programs which access *GRAMPS* facilities. Appendix C is a complete list of all the functions currently in the *GRAMPS* library. This is arranged by source file, and is intended for use by an advanced programmer or system administrator needing further insight into the workings of the operating system. It is suggested that the casual user skip this section. Appendix D describes the local development environment at the National Bureau of Standards, and Appendix E describes the environment at Philips Laboratories, and local modifications to *GRAMPS*.

---

[2] *Unix* is trademark of AT&T Bell Laboratories

The companion document *Administrator's Guide, [Ref. 2]* is intended for a system administrator. It explains how to set up common memory files and dynamic-memory free lists, and explains in some detail what happens during system initialization. The general user who simply wants to implement an application need not read that document.

User's Guide

A user of *GRAMPS* will typically want to run one or more processes in a stand-alone system. Usually, the processes will communicate with each other, and will run asynchronously (although synchronous operation is supported). This section describes how to write applications that make use of the *GRAMPS* primitives to develop such systems. For most operations, there are several different ways of achieving the same effect. In these cases, a preferred method will be described, and the options mentioned. In some cases, reference will be made to Appendix A without further elaboration. The following assumes the reader is familiar with the syntax of the *C* programming language.

An application to be run under *GRAMPS* must be manually broken into separate processes, since *GRAMPS* does not perform dynamic process allocation. Each process is coded as a separate program, in the usual way, with each process having a main program and whatever subroutines are needed. When processes need to communicate with each other, or with the outside world, *GRAMPS* function calls are used in the place of the analogous run-time library routines or operating system calls. For many such calls, the usual *C* syntax may be used (e.g., **read**, **write**). For two processes to communicate, it is necessary to set up one or more files. The programmers of the processes need only agree on a conventional file name (an ASCII string) that both will use to refer to each file , and use that name in opening the file [3]. This name should be given to the system administrator, together with an estimate of the maximum size (in bytes) of a single transaction. The system administrator will assign a common memory area for the file, and set up the appropriate initialization so that communication can be established.

Information about each of the *GRAMPS* files, including location and size, is stored in a disk file on the host computer, alongside the user's source code files. This file is given an extension of "**.cm**" by convention (for "common memory"). This file must be included in the main program of each application process (using **#include**). The system administrator will supply this file, and keep it up to date. Usually, the filename is the same as the process name (for example, a process called **p1** will have a main file called **p1.c**, and a **#include** file called **p1.cm**). A special "**cm**" directory will be established, where all the included files will be kept, so for the example above, the statement necessary to bring in the information about the files used by **p1** would be

#include <p1.cm>

The "**.cm**" files contain information that enables the process to find the common memory locations associated with each file name, and ensures that initializations happen in an orderly way. It should not be necessary to examine these files, but an example of one is given in the section for system administrators, together with explanations of all the entries.

Note that the file parameters may be overridden at run time, in that a special system process called SYS creates a system-wide list of files which are used in preference to the "**.cm**" entries. This assures that each process is using the same, current, file data. The "**.cm**" entries are primarily for use in debugging, when the SYS process may not be present.

Usually, a process running stand-alone will take the form of an endless loop, rather than running to a conclusion. Given the overhead of downloading a program, it is usually not cost-effective to execute a single operation on multiple processors. Thus, programs take a common form of polling their inputs, operating when they receive information, writing out their results, and repeating the process. Obviously, there can be many variations on this formula, but a common thread is the use of polling. *GRAMPS* as currently implemented does not support interrupt-driven I/O.

[3] In fact, the name does not have to be the same for the two processes, so long as the system administrator is told that the names refer to the same file. For simplicity, however, it is conventional to use the same name for all accesses to the same file.

The main features of the *GRAMPS* operating system will be introduced here by means of examples. Three annotated programs follow, two dealing with passing information by means of files, and one that makes use of the dynamic memory allocation capabilities. Included in the annotations are descriptions both of the *GRAMPS* system calls actually used in the programs, and alternatives that may be more appropriate in some situations. A listing of user-callable functions, with brief descriptions, appears in Appendix A. Both of the examples have been implemented and tested, using a pair of essentially identical programs, communicating with each other. Where several calls are available to perform similar or identical functions, the preferred version is noted. A convention throughout what follows is to use *italics* to describe variables and **boldface** for *GRAMPS* functions and *C* keywords.

Figure 1 shows a simple program for a process, **P1**, that writes data to a file, reads from another file, and prints what it read to the terminal. An analysis of this program will give a flavor of how *GRAMPS* programs are written. It is fruitful to study the program as a whole before reading the annotations. Note that, for the program to run properly, a second process, **P2**, will also need to be running. The second program in this case can be exactly the same as this one, except that it writes the file read by this program, and reads the one written by this program.

Line 1 — *p1.cm* contains information about the files and dynamic memory allocation areas specific to this process. It includes a structure (the *files* structure) that has an entry for each file giving its name, the address where its flags are stored for opening and closing the file, and the address and size of its data area. It should be included only once, in the main program. *p1.cm* is shown in Figure 1 in the *Administrators Guide*.

Line 2 — *vbus.h* contains structure definitions, external *GRAMPS* function declarations, and defined constants that may be useful to the user process. It also contains typedefs for **ADDR** (**char *** used as a pointer), **STR** (**char *** used specifically as a pointer to a string), **uchar**, **uint**, **ulong** (**unsigned char**, **int**, and **long**, respectively), and **USRBITS** (32-bit longword used as a set of user bit vectors). *vbus.h* may be **#include**'d in any file that makes use of its definitions. A listing is included in Appendix B.

Line 3 — NTRIES is a number indicating how many times to attempt to open a file before declaring it busy. This will be discussed in more detail below.

Line 4 — *user* is simply the name the programmer has given to the process (**P1** or **P2** for this example). This name must be given to the system administrator, and will be used as an identifier allowing processes to know who last used each file. User identifiers are one-byte numbers (**P1** is actually **#define**'d to be 0x01, in this case).

Line 6 — This is the beginning of the program. As required in *C*, the main program is called **main**. Note that there are no arguments to **main**. Since there is nowhere to pass arguments from, there is no way to give values to the conventional *argc*, *argv*, *env*.

Lines 8-9 — *fdin* and *fdout* are file descriptors for the two files. They can take values of -1, in the case of an error return from a file operation, FILEBUSY (defined in *vbus.h* to be -2), in the case of trying to access a file currently in use by another process, or a small positive number, when a file is open. They correspond almost exactly to the file descriptors usually available in *C*. *buf_in* is simply an array that will be used to store the data read from the input file.

Line 10 — This is the start of a never-ending loop. The program will execute the contents of this loop until a fatal error occurs, or the system is interrupted or reset by a user or another process. Most real-time programs will have such a loop.

Line 12 — This is the first call to a *GRAMPS* function, **flagpeek**. **flagpeek** returns the current value of the open/closed flag for the given filename. This flag consists of two bytes, one containing the status (open or closed) of the file, the other containing the ID of the current user (if open) or the previous user (if closed). CLOSEDTOYOU is a mnemonic **#define**'d in *vbus.h*. The **while** statement will loop in a null loop as long as YOU (this user) is the most recent user of the file. When any other user uses the file, the condition will no longer be met, and control will pass to the next statement.

```
1   #include <p1.cm>     /* the common memory map for this process */
2   #include <vbus.h>    /* the GRAMPS header file */
3   #define NTRIES  500          /* number of tries to open file */
4   extern unsigned int user; /* this process's ID, defined by the system */
5
6   main()
7   {
8   int fdin, fdout;  /* file descriptors for input and output */
9   char buf_in[20];              /* buffer for reading */
10  for(;;)                  /* loop forever */
11          {
12          while(flagpeek("sendtop2") == CLOSEDTOYOU) ;
13          fdout = openn("sendtop2", W, NTRIES);
14          switch(fdout)
15                  {
16                  case -1:  /* error in open */
17                          abort("ERROR in opening sendtop2\n");
18                  case FILEBUSY:       /* file is open to other process */
19                          printf("sendtop2 busy again\n");
20                          break;
21                  default:  /* file is now open to P1 */
22                          if(write(fdout, "1234567890", 11) != 11)
23                                  printf("bad write to P2\n");
24                          if(close(fdout) == -1)
25                                  abort("can't close sendtop2\n");
26                  }
27          fdin = openn_with_previous_user("getfromp2", R, NTRIES, P2);
28          switch(fdin)
29                  {
30                  case -1:  /* error in open */
31                          abort("error in opening getfromp2\n");
32                  case FILEBUSY:       /* file is open to other process */
33                          printf("getfromp2 busy\n");
34                          break;
35                  default:  /* file is now open to P1 */
36                                  /* there are new data in the file */
37                          if(read(fdin, buf_in, 11) != 11)
38                                  printf("bad read from getfromp2\n");
39                          else    printf("buf_in = %s\n",buf_in);
40                          if(close(fdin) == -1)
41                                  abort("can't close getfromp2\n");
42                  }
43          }
44  }
```

Figure 1.

It is preferable to use **flagpeek** to determine the *previous_user* flag, rather than repeatedly opening and closing the file. This is because opening and closing the file may interfere with another user trying to gain access, delaying him and possibly locking him out entirely. Also, opening the file causes this user's ID to be put in the *previous_user* flag, and the current time to be entered in the flag buffer, giving an incorrect picture of the status of the file to other users who may require this information, and to the debugging tools.

Line 13

openn is the basic file opening command, used here to open the file named "send-top2" for writing. It takes three arguments, the first of which is the file name, a string. The second is an **int**, which may be either *(int) 'W'* for read/write privileges, or *(int) 'R'* for read-only privileges. The names *W* and *R* have been **#define**'d to be *(int) 'W'* and *(int) 'R'*, respectively, in the header file **vbus.h** (Appendix B). (A **char** argument would not be portable and would lead to inconsistencies with *Unix System V*.) Note also that this argument is not a string, i.e., "W" (with double-quotes) would NOT be correct.

In any case, the second argument is meaningful only for *multiple-reader* files. A multiple-reader file is one that may be simultaneously accessed by several read-only users. Only one user may write the file at a time, however, and no other users may have access to the file during a write operation. The *W* argument will cause new users to be shut out while waiting until all current users have finished with the file, and only then allowing the file to be opened for writing. (Changing a single **#define** in *GRAMPS* allows multiple-reader files to be treated as ordinary files, if this is desired.)

The third argument to **openn** is the number of times the function will attempt to access a busy file before returning FILEBUSY. When this function is executed, it looks for a file in the *files* array with the given name. If it does not exist, or if the associated flags have been corrupted, **openn** returns -1. Otherwise, the file may already be open to another process, in which case **openn** tries up to NTRIES times before returning FILEBUSY. (If NTRIES is 0, it will continue trying as long as necessary.) Usually, only a small number of tries should be needed to open a file, unless some process is not behaving properly. If neither of these cases is found, the file will be opened, the identifier of the current process will be stored with the file as its current user, and a small integer will be returned as the value of the function call. The file will now be available for use by this processor. The following function calls are variations on this basic theme.

openn_fd(*fd*, *rwflags*, *ntries*) provides a much faster access to the file, by avoiding the need to look up the file name each time. It can only be used after the file descriptor, *fd*, has been established, which requires at least one call to **openn**, one of the other open calls, or **get_fd** (see Appendix A and Example 3). By obtaining the file descriptors for each file before the forever loop, the calls within the loop can be made more efficient.

open(*filename*, *rwflags*) is included for compatibility with *Unix*. It will attempt to open the file given by the string *filename*, but will keep trying if the file is busy (equivalent to **openn**(*filename*, *rwflags*, 0)). The *rwflags* argument may take the predefined values *W* or *R* (see **openn** above), or the *Unix System V* values *O_WRONLY*, *O_RDWR*, or *O_RDONLY*.

open_synch(*filename*, *rwflags*) is used when processes want to use synchronous communications. It will wait for the next tick on a system clock before opening the file. (This function will retry as long as necessary, since in normal usage different users will access the file on different offsets from the clock tick.) The ticks on the clock are defined as being a given offset from a given multiple of the basic system clock (see Appendix A). The offset and multiple are contained in the global variables *synchr_base* and *synchr_incr*, which may be set by the user.

openn_with_previous_user(*fd*, *rwflags*, *ntries*, *previous_user*) insists that a particular user was the last to use the file. Often, two processes use a file, but each process is only interested in reading what the other one wrote. This call can be used instead of **flagpeek** to ensure that the file is not continually being opened by the process that wrote the last data, only to be closed again immediately. The use of this call is illustrated in line 27 of the program in Figure 1. Note that at initialization a writer of the file will be permitted to **openn_with_previous_user** even though there is no previous user.

openn_with_otheruser(*fd*, *rwflags*, *ntries*) is similar. In this case the process waits until *any* user ID other than this user's appears in the flag. As soon as this happens, the file is opened.

Line 14 — There are three possible outcomes of an open call. The switch statement starting on this line accounts for each of them.

Lines 16-17 — If the open call resulted in an error, something is very wrong with the system. In this program, the **abort** system call is used to print a message and stop the process. Note that this is very drastic. A system with a supervisor might want to send an error message, try again, or request to be reset and restarted. Here the result will be that a trace of the functions called by the process will be printed on the terminal (if one is attached to the processor), and the program will stop. The **exit** call is an alternative. It does not print a message before stopping the process. If given an argument, **exit** will print a stack trace before calling **stop**, which halts the program.

Lines 18-20 — If FILEBUSY is returned, the file is currently being used by another process. That is, each time an attempt was made to open the file, it was found to be open to another processor (up to NTRIES times). Often this means that something is wrong with the other process, but this is not a fatal error, so a message is printed to the terminal, and the process continues. **printf** behaves exactly as it would in normal *C* implementations, except that it is specially written to check whether or not a terminal is attached to a processor. If there are many terminal output statements in a program (e.g., for debugging), the program will not run in real time, and the debugging may be invalid. Simply by disconnecting the terminal line from the board, however, it is possible to run at (almost) the normal speed. As soon as the terminal is reconnected, the message will reappear. This is sometimes the only way of determining why certain failures occur.

Line 21 — If neither of the first two cases has occurred, then the file has been opened successfully. (Note that we determined in line 12 that this process was not the last user of the file. Clearly this process has not used it since. Thus it is still not the last user, and no further examination of the flag is necessary. It is worth noting, however, that had we needed to know specifically that a particular user was the last user, it would NOT have been sufficient to have performed the **flagpeek**, since a different user may have opened the file after the **flagpeek** but before this user's **openn**. In this case a **lastuser**(*fd*) is required, once the file has been opened.) Since another user has opened (and since closed) the file, the previous data have presumably been read, so new information can be written. Thus, line 22 is executed. In this program, nothing is written until the old data have been read. This is not, however, enforced by the operating system. Many applications, such as real-time sensing, are better off overwriting old data before it is read, to ensure that the receiving process always has up-to-date information.

Lines 22-23 — The **write** call actually copies the data out to the file. It is analogous to the usual *C* write, and returns the number of characters written. Hence, the check for the number of characters, and the message if something goes wrong. As with **open**, there are a number of variations on the **write** command.

writeran(*fd*, *buffer*, *count*, *offset*) performs a random-access write, to an address *offset* bytes from the beginning of the file. Note that there is no **seek** command (although one could easily be implemented).

writepointer(*filename*, *wronguser*, *pointer*) (formerly called **writeheader**) is used when a single address pointer is to be passed to another process. It waits until some process other than *wronguser* was the previous user of the file, and then writes the address (in *pointer*) to the file. It does not need a prior **open** call, nor a following **close**. A common use is to create a structure containing pointers to all the dynamically allocated areas to be passed to another process. The address of this structure is

passed to the other process, which can then access all the areas.

autowrite(*filename*, *buffer*, *count*) performs an **openn** on the file, writes the data in *buffer* to it, and closes the file. It is similar to **writepointer**, except that it writes an arbitrary length buffer. Figure 2 shows its use.

Lines 24-25 After a file is opened, no other user can gain access until it has been closed. The **close** call changes the semaphore associated with the file to indicate that the file is available. The *previous_user* flag is not changed; it continues to show the process that last opened the file, and that is now closing it. Note that for the error (-1) and the FILEBUSY cases in the **switch** statement, the file was not successfully opened, so should not be closed. A call to **close** in these cases will result in an error message. An error in closing, signalled by a returned value of -1, is serious when the file was successfully opened. In this program, it will result in stopping the process with a message and a stack trace.

Line 27 Having finished with the write part of the program (whether or not it was successful), the next step is to read data from process **P2**, using the file called "getfromp2". As described above, the **openn_with_previous_user** call is used to ensure that **P2** has actually written new information to the file before the file is opened. The file is opened only for reading in this case, and NTRIES attempts will be made to open before FILEBUSY will be returned.

Lines 28-42 Just as in the previous open, there are three possible outcomes of the open call. The switch statement is analogous to the one described above, except for the **default** case (lines 35-41). If the file is successfully opened, it is guaranteed that **P2** was the previous user, so no special check need be done. The **read** statement is the same as the usual *C* statement, returning the number of characters read. Usually, the number of characters to read is known. It is possible, however, to keep on reading until no characters are received (or less than were asked for). This is not an error, unlike writing fewer characters than requested. A serious error will return -1. As for the **write** command, there are variations on the **read**.

readran(*fd*, *buffer*, *count*, *offset*) is analogous to **writeran**.

readpointer(*filename*, *wronguser*) (formerly called **readheader**) is analogous to **writepointer** (except that **readpointer** returns as its value the pointer read from the file). The declaration of **readpointer** as a function returning a pointer to characters appears in *vbus.h*. An example of the use of **readpointer** is illustrated in Figure 3.

autoread(*filename*, *buffer*, *count*) is analogous to **autowrite**. It opens the file, reads the specified number of bytes, and closes the file.

Lines 40-41 Once again, the **close** statement makes the file available to other users, and is only used on a successfully opened file.

The above example can be programmed more succinctly using **autoread** and **autowrite**. This is shown in Figure 2, which is the same basic program as in Figure 1.

Lines 6-7 Note that we no longer need the file descriptors. This bookkeeping is taken care of by **autoread/autowrite**. The variable *nchars* is introduced here only to show that these functions return a value, the number of characters read/written. *nchars* is not used in this program.

Line 10 As in line 12 of Figure 1, this line causes the process to loop, waiting until someone else uses the file.

Line 11 This line replaces lines 13-26 of Figure 1. Most of the functions done explicitly in Figure 1 are performed by the *GRAMPS* functions **autoread** and **autowrite**. An error in opening the file will result in a diagnostic message and exit from the program, with a stack trace. The same holds for errors in **read**, **write**, or **close**. Thus, there is no return of -1 possible. FILEBUSY conditions will result in looping within the **auto** subroutine, until the file is closed. This contrasts with the case in Figure 1, where

```
1   #include <p1.cm>        /* the common memory map for this process */
2   #include <vbus.h>       /* the GRAMPS header file */
3   extern unsigned int user;
4   main()
5   {
6   int nchars;                         /* number of characters read or written */
7   char buf_in[20];          /* buffer for reading */
8   for(;;)                   /* loop forever */
9         {
10        while(flagpeek("sendtop2") == CLOSEDTOYOU)) ;
11        nchars = autowrite("sendtop2","1234567890",11);
12        while(flagpeek("getfromp2") != CLOSFLAG(P2)) ;
13                  /* now there are new data in the file */
14        nchars = autoread("getfromp2",buf_in,11);
15        printf("buf_in = %s\n",buf_in);
16        }
17  }
```

Figure 2.

after NTRIES attempts at opening the file, control is returned to the user's program. The advantage here is that the user need not concern himself with programming to handle error conditions. Of course, the processor is halted on an error condition, which may sometimes be a disadvantage.

Line 12 This line is analogous to line 10. Whereas there we waited for any flag other than CLOSEDTOYOU, here we insist on a specific flag, closed-to-**P2**. **CLOSFLAG**(*arg*) is a preprocessor macro, defined in *vbus.h*, which constructs the flag meaning "closed to *arg*".

Line 14 Analogous to line 11. This line and the next replace lines 27-42 of Figure 1, but not exactly. **openn_with_previous_user** guarantees that if the file is opened, the previous user will have been the one requested. Here, the **while** loop guarantees the same thing, but a different user may have opened the file after the **flagpeek** but before the **open**. Thus a different user would in fact be the previous one. For files which are used by only two processes, as here, this cannot happen.

The above examples have illustrated many of the functions used to pass information between processes using files, or fixed, named, regions of common memory. The next example expands further on some of these methods, and introduces the use of shared, dynamically allocated memory. Here, one process allocates a region of memory, and passes pointers to the region to one or more other processes. Each of the processes makes use of the region as if it were part of its local memory. Processes may read and write at will. Usually, this can be managed so that there are no conflicts. If conflicts are hazardous, semaphores can be used for dynamically allocated memory, in the same way as for ordinary files, but the resulting slowing of access may make it more practical to copy the data to each process that will use it.

As processes finish with dynamically allocated regions, they must explicitly free them. The memory is not, however, returned to the free list until the last processor that has access to it has finished with it. Each processor to which the address is to be passed must be named in the call to the dynamic memory allocator, or added to or deleted from the list of eligible users at some later time. These requirements are elaborated below.

Figure 3 shows a program for a process, **P1**, that allocates common memory for a structure, opens a file, and passes the address of the common memory to another process, **P2**. **P1** then reads a pointer to dynamically allocated memory from **P2**, frees the memory for both regions, and exits. While this is

not a useful program, it illustrates the mechanics of the dynamic memory allocation and freeing process. Once again, the program should be read as a whole before the annotations are studied.

```
1  #include <p1.cm>        /* the common memory map for this process */
2  #define NTRIES  500     /* number of tries to open file */
3  extern unsigned int user;
4  main()
5  {
6  int fdout;             /* file descriptor for output */
7  char *mymem, *hismem;  /* pointers to memory for P1 and P2 */
8  fdout = get_fd("sendtop2"); /* get the file descriptor for output */
9  mymem = alloccm(20, P1BIT | P2BIT);
10 if(mymem == NULL)
11     abort("can't allocate common memory for mymem\n");
12 strcpy(mymem,"message for P2\n"); /* insert message in allocated area */
13 fdout = openn_fd(fdout, W, NTRIES);
14 switch(fdout)
15     {
16     case -1:   /* error in open */
17           abort("ERROR in opening sendtop2\n");
18     case FILEBUSY: /* file is open to another process */
19           printf("sendtop2 busy\n");
20           break;
21     default:       /* file is now open to P1 */
22           if(lastuser(fdout) != user)
23                 {
24                 if(write(fdout,&mymem,PTRSIZE) != PTRSIZE)
25                       printf("bad write to P2\n");
26                 }
27            else printf("this user still is previous user0);
28           if(close(fdout) == -1)
29                 abort("can't close sendtop2\n");
30     }
31 if(freecm(mymem) < 0)
32     abort("cannot free common memory for P1\n");
33 hismem = readpointer("getfromp2",P1);
34 if(hismem == NULL)
35     abort("could not read pointer from P2\n");
36 printf("message from P2: %s\n",hismem);
37 if(freecm(hismem) < 0)
38     abort("can't free common memory for P2\n");
39 stop();
40 }
```

Figure 3.

Lines 1-5 These are the same as for the program in Figure 1.

Lines 6-7 Here the variables used in the program are declared. Only one file descriptor is needed, and there are two pointers. The pointers will be used to address the areas in common memory. *mymem* will be allocated by this process, and sent to **P2**, while *hismem* will be allocated by **P2**, and read by **P1**.

Line 8 **get_fd** finds the file descriptor associated with the given file name, and returns it for later use. Making use of the file descriptor and the calls **openn_fd** and **flagpeek_fd** ensures that file accesses will be substantially faster than using the filename and **openn** (etc.) directly.

Line 9 — **alloccm** is the statement that allocates an area in common memory. The command here is to allocate 20 bytes of common memory, to be used by processes **P1** and **P2**. P1BIT and P2BIT are defined in *custom.h*, within *vbus.h*. The **alloccm** call returns a pointer to the start of the available space (a **char ***), which may be coerced to other types of pointers as necessary. Note that different processes allocate space in different areas within common memory, so that it is possible for one process to run out of space while others still have adequate space. The system administrator can reassign these areas if necessary.

Lines 10-11 — If there is some problem with allocating memory, or if there is no more memory available, the **alloccm** call returns NULL. This can be used as a signal for the user's program to free memory that is no longer needed. Here, the program prints a message and halts.

Line 12 — The pointer returned by the **alloccm** call can be used exactly like any other pointer in the system. Here, a string is copied into the area, to be read by **P2**.

Line 13 — Having found the file descriptor by calling **get_fd**, the **openn_fd** call can now be used to open the file. This works exactly the same as the **openn** call described in the first example. It is faster because it does not have to look up the file descriptor that corresponds to the file name. The same three results can occur after making the call as for the **openn** command in Example 1.

Lines 14-30 — The **switch** statement accounts for the three values returned from the open. There are no differences between this code and that described in Figure 1 except in the default case. Here, the pointer is written out to the file "sendtop2". Note that the *value* of *mymem* is written out to **P2**, by passing the *address* of *mymem* to **write**, as in *Unix*. Note that **P2** needs to know where in memory **P1** allocated the space, and this is the *value* of *mymem*, (and not, for example its address). As in Example 1, the file is closed after writing, to make it available to **P2**.

Lines 31-32 — After finishing with dynamically allocated memory, it is necessary to return it to the free list. The **freecm** command is the usual way of doing this. **freecm** removes the process from the list of legal users of the location, and, if there are no remaining legal users, returns the memory to the free list for re-use. Usually it is a serious error for **freecm** to return a negative value. Either an invalid address has been given as an argument, or the free list has been corrupted, or the memory has already been freed. Here, the process is halted if an error occurs. More robust systems would have to deal with the problem in some other way.

A more general form of **freecm** is **addusers**. **addusers**(*pointer*, *addvec*, *subvec*) takes a pointer to a common memory location, a vector of users to be added as having legal access to the location, and a vector to be removed from access to the location. **addusers** is used when the set of originally assigned users is to be modified. For example, **alloccm** always assumes that the allocator will be a user of the area. Often, one process simply creates and initializes an area, and then passes it to another process, without further use. The first process can remove itself from consideration as a user by calling **freecm** or **addusers** immediately after the location is allocated. Another use is when a process needs to pass a pointer to a third process unknown to the process that allocated the memory. The intermediate process would need to add the third process to the list of legal users before passing the pointer, or there would be a risk that the memory would be freed before it was claimed by that process. Note that when no more users have a claim to the memory **addusers** returns the area to the free list just like **freecm**, and the area can thereafter be reallocated at any time.

Lines 33-35 — Having written the address of the area allocated by **P1**, the example program shows one way of reading a similar address allocated by another process, in this case **P2**. **readpointer** was discussed above, and is used here to return the value of a pointer. (**readpointer** is declared to be a function returning (**char ***) in *vbus.h*.) The file used

for reading is "getfromp2", which is only read after **P1** is NOT the last user. That is, the file flags are checked until some process other than **P1** was the previous user, and a number of bytes equal to the length of an address is read and returned. If there is a problem with the read, **readpointer** returns NULL, which is used here to abort the process. Note that the above line 6, and most of lines 13 to 30 could have been replaced by a single **writepointer** call, which writes an address to a given file, as described above.

Line 36 — Having read the pointer from **P2**, **P1** may now use it for its own purposes, just like any other pointer. Here, the message is simply printed to the terminal.

Lines 37-38 — When the memory is no longer needed, it must be returned to the free list through the **freecm** call.

Line 39 — The **stop** statement causes an immediate halt of the program.

It should be pointed out that there are two types of dynamic allocation and freeing in the *GRAMPS* system. In addition to the common memory handling described above, local, non-shared memory can also be dynamically allocated. This is done using the usual *C* **malloc** and **free** calls. Memory will be allocated and freed in the same way as for common memory, but will only be accessible to the process that created it (and possibly other processes that run on the same processor). Even though the addresses of local pointers can be passed to other processes (as can any value), attempts to access these addresses will result in reaching areas within the local memory of the receiving process, rather than of the process that sent the pointer. This is usually disastrous.

In addition to these functions, there are a number of miscellaneous calls that have not yet been described. They can be divided into functions for dealing with files, for communicating with a terminal, and for accessing registers and memory locations. There are also system calls for process switching and synchronization. These are described briefly below.

Three system functions are provided for checking the flags associated with a file, without disturbing them, and without preventing other processes from accessing the files, as discussed earlier. The three calls are **flagpeek**(*filename*), **flagpeek_fd**(*fd*), and **lastuser**(*fd*). The first of these returns the flagword associated with the given file when given the file name, the second returns the flagword when given the file descriptor (which is faster), while the third returns only the byte containing the user ID of the previous user, and it too requires the file descriptor argument. The flagwords consist of two bytes, one containing the identifier of the previous user of the file (which may be the current user if the file is open). The second byte is used to determine if the file is OPEN or CLOSED. Successive calls to **flagpeek** or **flagpeek_fd** may give different values if the files are in use.

Note that the decomposition of the flagword into the two bytes is machine dependent. It is therefore recommended that this be done using the following preprocessor macros, which are defined in *vbus.h*. **FLAGBYTE**(*flagword*) selects the OPEN/CLOSED byte, and **PREVUSER**(*flagword*) selects the *previous_user* byte. Also **OPENFLAG**(*userid*) constructs the flagword that means 'OPEN to *userid*', and similarly **CLOSFLAG**(*userid*) for 'CLOSED to *userid*'.

Some of the functions that deal with terminal interaction have already been mentioned. In addition to the usual **printf**, **scanf**, **getchar**, **putchar**, etc., there is a function that reads from the terminal up to, and including the next newline, and returns the characters in a buffer. **readnl**(*dummy*, *buffer*, *count*) behaves like a normal read, except that it always reads a complete line of text from the terminal. Another useful function for terminal input is called **ifchar**() . This simply checks to see if something has been typed at the terminal. If not, 0 is returned. If so, the character is returned, however the character also remains in the buffer until read by a **getchar**. (**getchar**, by comparison, does not return at all until a character has been typed.) Note that **ifchar** will detect a ctrl-C, and stop the process. Finally, there is a call, **testcrt**() , that checks to see if a terminal is connected to the board. If so, it returns non-zero. This is useful in cases when timing is critical. The call is used in **printf** to ensure that processing is done only when there is something to receive the message to be printed.

There is also a class of calls that deals with timing in the system. Some of these assume that there is a clock available, so may not work in all cases. **sleep**(*n*) is used to wait some specified number of milliseconds (approximately) before doing some action. **sleep** does *not* require a system clock; it

obtains its delay by executing do-nothing loops. (The remaining timing functions *do* require a system clock.) **wait_until**(*t*) waits until the specified time, then returns the current time. If it is called when the given time has already passed, it returns immediately. **wait_to_multiple**(*t*, *dt*) waits until the time is a multiple of *dt*, after *t*, and then returns the current time. These calls can be used to simulate synchronous communications (see e.g. **open_synch**), or to ensure that an action, such as taking a picture, occurs at a given time. **systime**() returns the system time (a common memory counter incremented using a clock interrupt). **imalive**() sets a location in the current process's **PROC** array (see below) to the current system time, so that another (supervisory) process can know the current process is still running. **imalive** also returns the current time.

A number of functions are provided for accessing registers and memory locations directly. These are useful when interacting with special devices. **peek**(*addr*) returns the long word (32 bits) at the address *addr*, which must be even (this restriction is imposed by the hardware), while **peekb**(*addr*) fetches the single byte at the address *addr* (even or odd). Similarly, **regpeek**(*register*) returns the long word value in the specified register. Registers are specified as follows (for the *Motorola 680x0*). Arguments are integers, from 0 to 15, with 0 corresponding to *d0*, 1 to *d1*, etc., through 7 to *d7*; then 8 corresponds to *a0*, 9 to *a1*, etc., through 15 to *a7*. There is no command to modify registers at present. In analogy to the peek commands, there are poke commands. **pokel**(*addr*, *longword*) copies the value *longword* (32 bits) to the given address *addr*, which must be even. **pokew**(*addr*, *word*) does the same for word length data (16 bits), and **pokeb**(*addr*, *byte*) copies a byte (8 bits) to any given address *addr* (even or odd).

Other commands can be used to copy data rapidly (using the *loop mode* of the *68010*), or to zero blocks of memory. **movblk**(*source*, *destination*, *count*) moves *count* bytes (*count* < 65536) from *source* to *destination*. The source and destination must not overlap. **movblkfast**(*source*, *destination*, *count*) is similar but faster, requiring that the addresses be even, and the count be a multiple of four. **movbigfast**(*source*, *destination*, *bigcount*) has the same restrictions, except that *bigcount* is a long word, so up to four gigabytes can be moved. To zero a block of memory, use **zeroblk**(*address*, *count*) where *count* is the number of bytes (< 65536) to be set to zero, starting at *address*.

A large number of *GRAMPS* debugging tools are available. Some of these are automatically invoked by the programs, such as the stack trace on exit, while others may be explicitly requested to help locate programming errors. These are described in the following section.

Debugging Tools

Before discussing explicitly invoked debugging tools, we call attention to the design criterion that whenever the system detects an error, it writes a message to the terminal (if there is one). This message includes the name of the subroutine that detected the error and a brief description of the error. This practice can save the programmer from having to code print statements at each place where a -1 error return can occur. (The system printout can be suppressed at any time, if desired, by setting the global variable *sysdebug* to zero.) These messages may be further augmented by setting bit 1 of the **PROC** array's *debugflag* (see Selection 6 below). This bit causes a stack trace to be generated and printed with every system message.

Once a program is debugged, it can be made much smaller by relinking it to a *GRAMPS* library that has been created with the **#include** DEBUG switch (in *vbus.h*) removed. This removes many of the run-time checks and associated error message texts.

Another execution time debug feature is the "system subroutine trace" facility. For every major system function entered, a line is printed giving the name and arguments of the function. This feature is invoked by setting bit 0 of the *debugflag* byte in the user board's **PROC** array (see below). Already mentioned are the stack traces which are automatically printed whenever **abort** or **exit** (with a non-zero argument) is called.

The explicitly invoked **tools** are self-guiding and menu driven. **tools** is actually a standalone program (a C **main** program) which is created and downloaded by *GRAMPS* alongside the user's **main** program, but is never called by any part of the user's program. It uses a separate stack, and bypasses the monitor's register save area, in order not to destroy any information about the user's program under

test.

The *GRAMPS* initialization message includes a reminder to "type XG for tools" when in the *GRAMPS* monitor. Without this monitor, do a **GO** to location 21040 (hex), or find the location of the **tools** subroutine, and do a **GO** to it.

The main **tools** menu then appears, as shown in Figure 4. These menu choices are elaborated on in the following paragraphs.

```
CR  - menu
1,t - stack trace
2   - display current stack
3   - display system flags
4   - display user's 'files' array
5,s - symbol table (addresses, or examine/change)

6   - set debug flags
7   - eliminate/restore subroutines
8   - display this user's PROC structure
g   - resume program
x   - restart program
e   - execute subroutine
^E  - exit
    (Note - ctrl-c will clobber the register save area)

Enter menu selection (CR for menu):
```

Figure 4.

Menu Selection 1 (or "t") presents a stack trace, the chain of nested function (subroutine) calls, most recent first. The addresses are obtained from the stack, which must not have been changed since the program was stopped. These are converted to the ASCII function names using the symbol table, which is downloaded by the download program at the same time as the code. The very last calls listed by the trace should be **main**, **vmain**, and **_main**. **vmain** and **_main** are *GRAMPS* initialization routines, and **main** is the user's main program. A stack trace run before starting the program is not meaningful.

Selection 2 displays the contents of the stack itself, or at least of the most recent 256 bytes. Note that the "top" of the stack, its most recent entries, are at the top of the display; however, these are the *lowest* stack addresses. The stack grows downwards, from higher addresses to lower.

Selection 3 is particularly useful. This shows the current state of the system file flags, i.e. which files are open and to whom, which have been closed and when (if the system clock has been running), and who last did a "peekflag". If a parameter file ("parmfile") has been created by the system, then all files will be listed (use control-S/control-Q to stop/start the output). Otherwise, only those files known to the process, i.e. appearing in its **.cm** file, are listed.

The user's internal *files* array is listed with Selection 4. This shows *this* user's (the user executing **tools**) own idea of the state of the file. Be careful, however, since some of this information has a slightly different meaning here than in selection 3, and moreover is updated only when the user attempts to open the file. Thus the open/closed flag tells only whether the file is open to *this* user, not whether it's open to any *other* user. Also, the *previous_user* flag refers to the previous user at the time this user last opened the file, and *not* the *current* user. In particular, if the file is open to *this* user, it is still the *previous* user that appears.

The symbol table, Selection 5, or "s", is probably the most important of the tools. Of course, a symbol table must have been downloaded along with the program. The symbol table is in a special compact format, having been created from the compiler's output by a *GRAMPS* formatter program **binmap**. The Administrator has probably put a call to this program into the shell file **llf** ("link, locate, and format"), so that it is created automatically.

On entering Selection 5, another menu appears, shown in Figure 5. Note that one may return to the main **tools** menu at any time by typing control-T (for **t**ools), or directly to the monitor by typing control-E.

```
Enter a global symbol name, or an address, or CR for menu.
To examine or modify a variable, enter its symbol or address followed
by %spec, where %spec is one of the following printf specifications:

  %d   int  (output in decimal)
  %ld  long (decimal)
  %x   int  (hex)
  %lx  long (hex)
  %c   char (ASCII)
  %cx  char (hex)
  %s   string (ASCII)
  %&s  pointer to string (ASCII)
  %f   float
  %lf  double
  %m   memory display (256 bytes)
  %a   array or structure (display sizeof(array) bytes)

You may also enter CTRL-T to return to the Tools menu,
 or CTRL-E to Exit directly to the monitor.

Enter a global symbol name, or an address (or CR for menu):
```

Figure 5.

The simplest function is just symbol table lookup. Enter a symbol name, followed by carriage return, and the program responds by printing "address = ", followed by the address. Conversely, enter an address (a hexadecimal number up to 8 digits), and the program prints the name of the global symbol at that address. If there is no symbol with that address, the closest symbol with a *lower* address is printed. Thus, if an address within a function is given, the name of that function is printed.

To examine or modify a global variable, it is only necessary to enter its name or its address as above, followed by a **printf** specification (intervening blanks are permitted). For example, to print the long (32-bit) integer variable *var* in hexadecimal, one would enter

```
var%lx
```

The specification is required because the symbol table does not store information on the nature of the variable: its size, or the interpretation of its bits (integer, string, floating point). The specifications listed in Figure 5 are accepted. Most of these mean the same here as in **printf**, and are thus self-explanatory. A few deserve special mention.

The %c specification, as in **printf**, prints the character value of the addressed byte. The character "A" prints as "A". But a non-printing character like control-G is, well, non-printing. The %cx has been added to print the two-digit hexadecimal value of the addressed byte. Note that hardware constraints of the *680x0* cause an exception (trap) if a **word** or **long** specification is requested at an odd address, so it

is necessary to use %cx for byte values.

A tougher distinction needs to be made between %s and %&s. %s is used when the address given, or the address of the symbol given, is the address of the first character of the string. For instance if *name* is declared by **char** *name*[5] = "ABCD", then the entry *name %s* would get the respone "value = "ABCD"". But when the address or symbol given is that of a pointer to the string, the other specification %&s needs to be used. Thus if we define **char** **pointer_to_name*, and we set *pointer_to_name = name* then the entry *pointer_to_name %s* gives garbage, but *pointer_to_name %&s* again gives "value = "ABCD"". This is because the object at address *name* (= *&name*[0]) is the first character of the string, while the object at *&pointer_to_name* is not a character at all but another pointer, to the actual string. In practice one may try both specs.

Finally the specifications %m and %a have been added to deal with arrays, structures, and general memory dumps. %m gets a dump of 256 bytes starting at the given address or symbol. Similarly, %a gives a dump, but to the end of the array or structure. This end is determined from the symbol table by scanning for the next higher addressed symbol. If the symbol and the next higher symbol are not contiguous--perhaps there are compiler-generated constants in between, for example, or a structure is aligned to a word boundary-- this algorithm will yield extra dump. For %m and %a specs, no modification of the variables is allowed. But one can find the address of, say, a structure element of interest, and then modify it using its numerical address and another of the specs.

Following the "value = " display just described, the program will prompt with "New value:". Continuing the example above ("var%lx"), the computer might respond with

value = 1A549D. New value:

A carriage return entered in response means "Make no change". Otherwise, enter the desired new value *in the same format* implied by the **printf** specification just entered (hexadecimal, in this example).

Only global symbols are recognized. Function names are inherently global. Variable names declared outside any function are also global (and thus anything referenced as **extern** is also). But variables declared within a function are **automatic**; these appear and reappear in various places on the stack, and are not included in the current symbol table.

Returning to the main menu (Figure 4), item 6 allows the user to view and change the *debugflag* byte in the **PROC** array. (See Selection 8 for a discussion of the **PROC** array.) The system currently recognizes bit 0 ("system subroutine trace": print name and arguments on entry to the major system functions) and bit 1 (print stack trace along with system error messages). Of the remaining bits, bits 2 and 3 should be reserved for future *GRAMPS* debug tools, while bits 4 through 7 are available for user use, for example to control the user's own debug printouts. The global variable *_debugflag* is set to the **PROC** array's *debugflag* at program start and occasionally thereafter (currently at each call to **dprintf**). There are two other ways to change the *debugflag* byte. One is "by hand", using the monitor on any board (since the **PROC** array is in common memory). The other way is by giving an argument "-dxx" to the downloader; where *xx* is the desired value (in hex) of *debugflag*.

Selection 7, eliminate or restore subroutines (functions), is often a quick way to determine which function is the site of a bug. It will request the name of the function, and whether to eliminate or restore it. Elimination involves placing a *return-from-subroutine* command as the first command to be executed; restoring is simply restoring the *link A6* with which it started. All C programs compiled with the Intermetrics compiler, most *GRAMPS* assembly-language subroutines, and very likely many other compilers' functions start with *link A6*. The **tools** program will not eliminate anything not beginning with *link A6*.

Selection 8 displays the user's **PROC** structure (its entry in the **PROC array**) in a readable fashion. The **PROC** array is an array of structures, one for each processor, which resides in common memory (at E20C00 in the NBS vision system). The address of a process's entry in the **PROC** array is saved on its own board at location THISPROC, at 20600 hex (in the current NBS implementation), and in the global variable *thisproc* (declared as **struct PROC ***) which is available to the user's program and thus is also in the symbol table. The **PROC** structure itself is defined in *vbus.h* (Appendix B). It is

32 bytes long, and includes the user ID, program status (initializing, running, or aborted), the *debugflag* described in Selection 6 above, the user's time (see **imalive**()), program start address, location of data segment, and PROM entries. The PROM entries will not be meaningful when used with other than *GRAMPS* PROMs. Examining the entire **PROC** array (*i.e.* for all processes) may be useful in getting a snapshot of the whole system.

Finally, single-character codes allow various ways of exiting **tools**. *Control-E* will exit to the monitor from anyplace where input is expected, without writing into the register save area. (*Control-C*, available in the user's program, will rewrite the save area, thereby destroying data regarding the program under test, and should not be used from **tools**.) *g* will goto (resume at) the place the user's program was last stopped. It is not recommended to use *g* if the user's program was never started. *x* will restart the user's main program (including the *GRAMPS* prologue) from the beginning. *e* will prompt for the name of a subroutine to be executed, invoke the subroutine, and return to **tools**. This is currently implemented only for subroutines with no arguments.

Appendix A

This appendix lists the functions in the *GRAMPS* library which are intended for use by user processes, with brief descriptions, taken where possible from the function code itself. Additional functions for system use are listed in Appendix C. Functions whose names are identical to the usual C functions have the usual C arguments also (**open**, **close**, **read**, **write**, etc).

Argument types, and types returned by functions, are (16-bit) **int** except where otherwise noted. Types **uint** and **ulong** are **unsigned int** and **unsigned long**, respectively; **ADDR** is **char *** and **USRBITS** is **unsigned long**. (These are defined in *vbus.h*)

Functions of the *GRAMPS* system that open or close files:

open(*filename*, *rwflags*) **STR** *filename*; Open a file. If file is busy, keep trying. Equivalent to **openn**(*filename*, *rwflags*, 0). The second argument is used only for multiple-reader files; in other cases it must be present, but its value is irrelevant. An ((**int**) 'W') as the second argument means write privilege is being requested; the *Unix System V* symbols *O_WRONLY* and *O_RDWR* are also recognized as write requests. Any other value gets read-only. The symbols *W* and *R* are defined in **vbus.h** to have the appropriate values. Note: "w" (the string) is NOT a valid write argument. This call is compatible with *Unix*.

openn(*filename*, *rwflags*, *ntries*) **STR** *filename*; **uint** *ntries*; Try to open a file *ntries* times (if busy). If there's an error, return(-1) immediately. If file is still busy after *ntries* tries, print a message and return(FILEBUSY) (FILEBUSY == -2). *ntries* == 0 means keep trying until open. The second argument, *rwflags*, is the same as for **open**. **openn** is preferable to **open** in finished systems, where a program which blocks indefinitely is not acceptable. The condition in which a file is busy for an unexpectedly long time requires corrective action by the affected process. However, for systems in a development phase **open** *is generally quicker to code (since there is no FILEBUSY return path), and is Unix*-compatible for easier testing.

openn_with_previous_user(*filename*, *rwflags*, *ntries*, *previous_user*) **STR** *filename*; Open a file, but wait until previous user is *previous_user*.

openn_with_otheruser(*filename*, *rwflags*, *ntries*) **STR** *filename*; Open a file, but wait until *previous_user* is other than this user.

openn_fd(*fd*, *rwflags*, *ntries*) **uint** *ntries*; Open a file, given its file descriptor (index into *files* array). This should be much faster than **openn**.

open_synch(*filename*, *rwflags*) **STR** *filename*; Wait until system clock is a multiple of *synchr_incr* offset from *synchr_base*, then open. This provides for synchronous communication, if desired. The variables *synchr_incr* and *synchr_base* are **extern unsigned int** and **extern unsigned long**, respectively, and may be changed by the user.

close(*fd*) Close the file with file descriptor *fd*.

flagpeek(*filename*) **STR** *filename*; Read the flag of the specified file without changing it.

flagpeek_fd(*fd*) Same, given the file-descriptor. Faster.

lastuser(*fd*) Returns the *previous_user* flag obtained from the system file flags. If the file is currently open to you, returns the user previous to you. (May need **get_fd**, below.)

get_fd(*filename*) **STR** *filename*; Get the file-descriptor of the named file. In *GRAMPS*, the file-descriptor is permanently associated with a named file, so this is meaningful even for an unopened file.

The following **preprocessor macros** are provided (in *vbus.h*) in order to insure portability of operations which form or decompose flagwords. This decomposition depends on the order in which bytes are concatenated to form words, which is different on an *8086* processor than on a *680x0*, for example.

FLAGBYTE(*flagword*) selects the byte that contains the OPEN/CLOSE information.

PREVUSER(*flagword*) selects the byte that contains the previous user's ID.

OPENFLAG(*userid*) constructs a flag word that says 'OPEN to *userid*'.

CLOSFLAG(*userid*) constructs a flag word that says 'CLOSED to *userid*'.

Read, write, and their analogs:

autoread(*filename*, *buffer*, *count*) **STR** *filename*; **ADDR** *buffer*;

autowrite(*filename*, *buffer*, *count*) **STR** *filename*; **ADDR** *buffer*; These two functions perform their own open and close.

read(*fd*, *buffer*, *count*) **ADDR** *buffer*;

write(*fd*, *buffer*, *count*) **ADDR** *buffer*;

readran(*fd*, *buffer*, *count*, *offset*) **ADDR** *buffer*; **long** *offset*; Direct access read ('readrandom') - read data starting *offset* bytes from the beginning of the file.

writeran(*fd*, *buffer*, *count*, *offset*) **ADDR** *buffer*; **long** *offset*;

ADDR readpointer(*filename*, *wronguser*) (formerly called **readheader**) **STR** *filename*; **char** *wronguser*; Check that someone OTHER than *wronguser* has used *filename* then read and return pointer (**ADDR**).

writepointer(*filename*, *wronguser*, *pointer*) (formerly called **writeheader**) **STR** *filename*; **char** *wronguser*; **ADDR** *pointer*; Check that a user other than *wronguser* has used previous pointer, then write new pointer.

Initialization:

vmain() Automatically called at startup by **_main**, to initialize the *GRAMPS* system. Should never be called by user.

filinit() Reset flags of all files owned or currently accessed by this processor. This function is automatically called at startup, by **vmain**. For any flag which is initialized, the remaining 14 bytes of the system's flagbuf are zeroed, and the entire file is zeroed also. Should never be called by the user.

Dynamic common-memory allocation programs:

Note the distinct names: **malloc** and **free** are the C-library routines that get space on the user's own board for its internal use; **alloccm** and **freecm** get space in common memory, which is accessible to other users.

ADDR alloccm(*size*, *uservec*) **uint** *size*; **USRBITS** *uservec*; Get a block of common memory, of the requested *size* (bytes). Return a pointer to it (**ADDR**). *uservec* is the bitwise *OR* of all userbits who will get use of this block.

freecm(*fp*) **ADDR** *p*; Return a block of common memory to the free list. **freecm** removes your userbit from the uservec; only if you are the last user does the block actually get returned to the free list.

addusers(*fp*, *addvec*, *subvec*) **ADDR** *fp*; **USRBITS** *addvec*, *subvec*; Update the bit vector of a block of memory by adding the bits in *addvec*, and removing those in *subvec* (which may include the current user himself).

Terminal I/O:

printf(*control*,*args*, ...) **STR** *control*; The usual C print-to-terminal function. However, printing is skipped if no terminal is present.

readnl(*dummy*, *buffer*, *count*) **ADDR** *buffer*; Read from terminal to (and including) the next newline. The returned count includes the newline (as on *Unix*).

getchar() returns(char). I/O primitive for reading from the terminal. Actually returns an int with high-order byte 0. Control-C will cause the process to stop (by calling **stop**) as soon as it is read in, i.e. at the next **getchar**, **putchar**, or **ifchar**).

putchar(*char*); I/O primitive for writing to the terminal. Control-S and control-Q may be used during output to turn transmission off and back on, respectively. (The program will block - pause - during this time.) Control-C will cause the process to stop (by calling **stop**) as soon as it is read in, i.e. at the next

getchar, **putchar**, or **ifchar**).

getch() returns(char). **getch** is like **getchar**, but does not echo the character to the terminal.

ifchar() returns(char). Returns 0 if no character is waiting to be read from the terminal. Otherwise, returns the character but doesn't 'use it up': the next call to **getchar** will return the same character. Normally used to see whether a character is waiting, if you don't want to block (hang); **getchar** will block if there is no character (as in *Unix*).

testcrt() **testcrt** is a subroutine to test if a device (presumably a CRT) is hooked up to the UART port. Returns nonzero if a device is there, 0 if not.

Stop and related functions:

exit(*i*) stop, and print a stack trace if *i* is non-zero.

abort(*fmt, args*) **STR** *fmt*; Stop, and print 'ABORT' and the specified message "*fmt*", filling in the args as in **printf**. Provide a stack trace.

stop() This is the simplest way to exit the program when something goes wrong, without doing a return to noplace.

Timer functions:

systime() returns(longword). Returns system time.

imalive() returns system time(long). Sets user time to systime to show SYS that this process is still alive.

sleep(*n*) *n* is approximately in milliseconds.

ulong wait_until(*t*) **ulong** *t*; Wait until the specified time, then return current time (unsigned long). If it's past the specified time, return immediately.

ulong wait_to_multiple(*t*, *dt*) **ulong** *t*; **uint** *dt*; Wait until time is a multiple of *dt* offset from *t*, then return current time (unsigned long).

Moves, peeks, etc:

movblk(*sourceaddr*, *destaddr*, *bytecount*); **ADDR** *sourceaddr*, *destaddr*; **uint** *bytecount*; does a block move (source and destination buffers must not overlap).

movblkfast(*sourceaddr*, *destaddr*, *bytecount*); **ADDR** *sourceaddr*, *destaddr*; **uint** *bytecount*; **movblkfast** is similar to **movblk**, but addresses MUST be even, and *count* MUST be a multiple of 4. There is NO checking. Count must be less than 65535.

movbigfast(*sourceaddr*, *destaddr*, *bytecount*); **ADDR** *sourceaddr*, *destaddr*; **ulong** *bytecount*; **movbigfast** is like **movblkfast**, except *count* is a long (up to 4 M). As in **movblkfast**, addresses MUST be even, and *count* MUST be a multiple of 4. There is NO checking.

zeroblk(*destaddr*, *bytecount*); **ADDR** *destaddr*; **uint** *bytecount*; **zeroblk** zeroes a block of memory.

peek(*address*); **ADDR** *address*; returns(longword); Fetches the longword at location *address*. *address* must be even (except *68020*).

peekb(*address*); **ADDR** *address*; returns(byte); Fetches the byte at location *address*. Note that **peek** cannot be slipped in in place of **peekb**, since the desired byte is returned at the other end of the longword return register.

pokel(*address*, *longword*); **ADDR** *address*; **long** *longword*; Stores a longword at location *address*. *address* must be even (except *68020*).

pokew(*address*, *word*); **ADDR** *address*; **uint** *word*; Stores a word at location *address*.

pokeb(*address*, *byte*); **ADDR** *address*; **char** *byte*; Stores a byte at location *address*.

regpeek(*register*); **int** *register*; returns(long); Returns contents of register (0-7 gets d0-d7; 8-15 gets a0-a7).

Miscellaneous functions:

setcurrpic(*n*) **uint** *n*; Set the currpic field of the **PROC** array (see Appendix B) to *n*.

changeproc(*userid*) **uint** *userid*; Change process identity (to *userid*) (but still on same processor).

STR index(*c*, *s*) **char** *c*; **char** **s*; Return a pointer to the first occurrence of *c* in string *s*; or NULL if none.

char makeprint(*c*) **char** *c*; Mask out the parity bit of the character argument *c*. If *c* is a printing character, return it; if *c* is non-printing, return '.' instead.

Debugging tools directly callable by user:

tools() A standalone program that may be entered directly from the monitor. Tools provides debugging tools for understanding the current state of the user's program. It takes care not to clobber the user's stack or register save area. The tools themselves are listed on and selected from a menu, *q.v.* Figure 4.

traceall() Print stack trace of the currently running program. This version is called by **exit** or **abort**, and so ends with a call to **stop**().

trace() Print stack trace of a stopped program. Uses saved_A6, in the register save area.

dispmem(*addr*, *n*) **ADDR** *addr*; Display *n* bytes of memory.

checkstack() Checks the current stack pointer to see if the stack is infringing on program space. If less than 4k bytes remain, the program aborts.

Multi-tasking functions:

relinquish() - issue a *trap 1* to voluntarily return control to the **scheduler**. This is the only user-callable multi-tasking function. Those functions callable from the (application-dependent) scheduler are listed in Appendix C.

Appendix B - listing of vbus.h and vbuscustom.h

```
/* <vbus.h> itself has those definitions which should not change */
/* from one installation to the next. Things which may change are kept in */
/* <vbuscustom.h>, which is #include'd automatically by */
/* vbusx.h A few definitions in the system header file may also */
/* change; these are listed also, at the end of this Appendix. */

/* Related header files: vbyssys.h, vbusparm.h */

/* .cm files may also have some #define's, and some variables */
/* defined. vbus.h includes remnants of stdio.h; DO NOT */
/* include that file too. */

#ifndef VBUS_DOT_H_IN          /* define only if we haven't defined before */

#define VBUS_DOT_H_IN          /* say vbus.h has been included already */

/* #define MULTITASKING 1      /* include this #define for multi-tasking */

    /* typedef's */

typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned int u16;
typedef unsigned long ulong;

typedef unsigned long USRBITS;      /* user bit vectors */
typedef char * ADDR;                /* any address */
typedef char * STR;                 /* strings in particular */

typedef struct alloheader * p_alloheader;

    /* structure definitions */

struct PROC {                       /* PROC (process) array on INTF board */
    uint useruser;          /* each byte has the value 'user' */
    uint status;        /* 0x1111   user program
                             0x2222   GRAMPS initialization
                             0x9999   task suspended
                             0xABAB   aborted
                             0xA0AB   interrupted (NMI)
                           */
#ifdef MULTITASKING
    struct context *context;
#else
    USRBITS usermask;
#endif
    uint currpic;           /* picture currently being worked on */
    char reserved[4];       /* The first 2 bytes, in INTF only,
                               are used as SYSINITFLAG */
    char debugflag;
    char irflag;            /* interrupt/reset flag */

    ulong time;
    ADDR start;
    ADDR data;
```

```
        u16   busyflag;
        u16   startopt;
        };

/*
        The filebuf structure is internal to the user's program.
        It contains the location of the named file, who owns it and
        who are the other users, and this program's idea of whether it
        has the file open and if so, who used it previously. The
        system, of course, uses only the 'official' status which is
        maintained in common memory, in the 'flagbuf's.
*/

struct filebuf {                    /* one structure for each file */
        char   filename[16];

        char   status;          /* open or closed (this flag is internal to program */
        uchar  previous_user;       /* obtained from flagaddr when file is opened */
        USRBITS otherusers;     /* Each bit is a user. Bit 31 set means this is the owner */
                                /*  (owner is responsible for resetting uninitialized flags at startup) */
                                /*  This user must NOT be among otherusers */
        uchar  multiple;        /* True says more than one reader at a time may access file */
        uchar  uninitflags;     /* bit 0 says if msg printed, bit 1 if uninitialized */
        uchar space[8];     /* filebuf should line up, i.e. be a multiple of 16 bytes */

        struct flagbuf *flagaddr;     /* */
        ADDR   buffaddr;        /* starting address of file */
        ADDR   endaddr;         /* last valid address plus one */
        ADDR   next;            /* address of next read or write position */
        };

struct flagbuf {                /* this lies in common memory */

        uchar  openflag;
        uchar  curruser;

        uint   opentime;        /* time file was last opened (low-order part) */
        uint   closlime;        /* time file was last closed */
        uint   peekuser;        /* time flag was last peeked at */
        USRBITS rdusers;            /* bit vector of users currently reading file */
                                /* (for multi-reader files only) */
        USRBITS writereq;           /* bit vector of user (if any) wanting (or */
                                /*  having) write permission. */
};

            /* for allocation blocks: */

struct alloheader {
        p_alloheader nextfree; /* pointer to (header of) next free block */
                                /* (for allocated blocks, points to itself) */
        p_alloheader hnext; /* pointer to (header of) next block (free or allocated) */
        ulong  hsize; /* the number of bytes in this block, including header */
        USRBITS  huservec; /* bit vector pointing to users of this block of memory */
        };

struct IDstruct {
        uint   id;
        STR    name;
        };
```

```
  /* basic #define's useful to users as well as system */

#define NULL    ((ADDR) 0)
#define EOF (-1)

#define TRUE    1
#define FALSE   0

#define LOBYTE  0x00FF
#define HIBYTE  0xFF00

#define OPEN    0x80            /* bit 7 set. This value is necessary in order
                                to be able to use the 68000's 'TAS' instruction */
#define CLOSED  0
#define FILEBUSY -2
#define UNINIT    2     /* do not have another user with this ID *//* why not ??? */

#define FLAGBYTE(flagword)   ((flagword) >> 8)
#define PREVUSER(flagword)   ((flagword) & LOBYTE)

#define OPENFLAG(userid)     ((userid) | (OPEN << 8))
#define CLOSFLAG(userid)     ((userid) | (CLOSED << 8))

#define OPENTOYOU            (OPENFLAG(user))
#define CLOSEDTOYOU                 (CLOSFLAG(user))

#define R               ((int) 'R'    /* read only (used in open, etc) */
#define W               ((int) 'W'    /* read-write (used in open, etc) */
#define O_RDONLY            0                   /* System V flags used in 'open' */
#define O_WRONLY            1
#define O_RDWR          2

/* defines used in userids */

#define ALLBITS  0x7FFFFFFF
#define OWNERBIT 0x80000000
#define INITBIT  0x80000000   /* used in openalloccm */

/* standard C file definitions: */

#define     stdin  0
#define     stdout 1
#define     stderr 1

/* alternate definitions for allocation functions */

#define     free(p)     cfree(p)
#define     malloc(n)   calloc(1,n)

/* definitions of commonly used GRAMPS functions returning other than int */

extern ulong peek(), peekl(), systime(), imalive(), regpeek();
extern      char  peekb();
extern ADDR       alloccm(), readpointer(), calloc(), fgets();
extern      double      atof();


#ifndef MAIN
extern struct IDstruct _userids[];
```

```
externstruct filebuf  *files;  /* the 'files' array is defined in *.cm */
#endif        /* end #ifndef MAIN */

#include <vbuscustom.h>  /* see below */

#endif/* end #ifndef VBUS_DOT_H_IN */



/*************************************************************************/

      /*    vbuscustom.h    */

/*************************************************************************/
/* <vbuscustom.h> has the #define's and initializations that may vary
   from one site to another */

      /* major system-wide parameters */

#define SYSTEM   1              /* NBS "lab" system. Mostly affects choice of
                                camera parameters */
#define FLAGBASE 0x400000  /* start of system's copy of common-memory I/O flags */
#define FILEBASE 0x401000  /* start of file (buffer) space */
#define ALLOBASE 0x460000  /* start of allocation area */

#define M     0             /* M=1 for MULTIPLE-reader files, M=0 for no */

#define MAXFILES 20           /* max # of files available to this process */
#define MAXPROCS 32           /* max # of processors. This should never exceed
                              32, because we run out of userbits */
#define THISPROC 0x20600      /* location of pointer to this process's
                              PROC structure. Set by PROM. */
#define DEF_SYNCHR_BASE 0/* default, for synchronous I/O */
#define DEF_SYNCHR_INCR 28        /* clock ticks, milliseconds (approx) */

   /* user id's: */

#define INIT 0x66  /* used to be 0. x66 is better, because flags won't
                      power up with that value */
#define UNSET     0x55  /* used in files.previous_user, which may be initialized
                      to INIT, or totally UNSET */
#define NONE    0xEE    /* used in userids table below */

#define FSV     0x10
#define SSVA    0x2A
#define MLDI    0x31
#define SYS 0x7F

   /* user bit numbers: */     /* (these are the indices, i.e. the number of bits
                              to be shifted: usermask = 1L << userbitn) */
#define FSVN  0
#define SSVAN 1
#define MLDIN 3
#define SYSN  14

   /* user bit masks: */     /* (these are the 16-bit masks, used for testing */
                           /* (userbit = 1L << USERN) */
#define FSVBIT  1
```

```
#define SSVABIT 2
#define MLDIBIT 8
#define SYSBIT  0x4000

#ifdef  MAIN
struct IDstruct _userids[MAXPROCS] = {
                /* arranged so that _userids[userbitn].id == user  */
                /* i.e., given a userbitnumber n, the user id  */
                /* is just _userids[n].id */
        FSV,  "FSV",
        SSVA,       "SSVA",
        NONE,       "",
        MLDI,       "MLDI",

        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       "",
        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       "",

        NONE,       "",
        NONE,       "",
        SYS,  "SYS",
        NONE,       "",

        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       "",
        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       "",
        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       "",
        NONE,       "",     NONE,       "",     NONE,       "",     NONE,       ""
};

#ifndef SOUP            /* The following is actually used only in SUP */
ADDR __allfiles;    /* dummy. */
#endif

#endif      /* end #ifdef MAIN */



/*************************************************************************/

        /*    from vbussys.h    */

/*************************************************************************/

/*-------------------- INSTALLATION-DEPENDENT CONSTANTS ------------------*/

#define ALLFILEMAX 50   /* Max number of files in entire system
                        (= allfiles array size) */
#define TIMEOUT 20000    /* max number of retries (opening a file) before printing warning */

/* #define MULTIPLE             /* commented out means don't include
                        multiple-reader file protocol */
#define SYSDEBUG            /* perform run-time checks */

        /* board locations */

#define USTART   (ADDR)0x21000    /* entry point to (GRAMPS prologue to) user's program */
#define UBASE    (struct baseblock *) USTART   /* start of user's baseblock, q.v. */
#define SYM_TAB_ADDR_SLOT       (struct symbol **) (UBASE->symbol_table_addr)
                        /* location of the symbol table address */
```

```
#define BOARDTOP ((ADDR) 0xA0000) /* highest address on board, plus 1 */
#define USERSTACK BOARDTOP     /* top of user stack space */
#define ZEROFROM (ADDR)0x90000 /* zero stack from here up to current stack pointer */

#define SYSINITFLAG    (((struct PROC *) 0xE21000) -> reserved)
                       /* in PROC array, INTF board's entry */

      /* The following are PROM addresses used only in tools() (vbusg.c),
         and will depend on the PROM used */

#define SGO        ((long *) 0x2022A) /* address of the address saved by */
                                      /* stop(), in monitor's save area. */
                                      /* Printed out by trace(). */
#define USRBUF   ((ADDR) 0x2022E)  /* "save area" - monitor's list of */
                                   /* register contents as of the last stop() */
                                   /* or breakpoint  */
#define reg_A6     regpeek(14) /* frame pointer register */
#define saved_A6   peekl(USRBUF + 14*4)   /* frame pointer register (in save area)*/
#define saved_A7   peekl(USRBUF + 15*4)   /* stack pointer register (in save area)*/
#define GO7 ((ADDR) 0x9BC)   /* address in monitor of end of 'G'
                                command */

/*--------------end of installation-dependent constants---------------*/
```

Appendix C

This appendix lists all the functions included in the *GRAMPS* library, under the source code file in which they appear, for use by the advanced user. Again, brief descriptions are taken where possible from the function code itself. Argument types, and types returned by functions, are (16-bit) **int** except where otherwise noted. As before, types **uint** and **ulong** are **unsigned int** and **unsigned long**, respectively; **ADDR** is **char *** and **USRBITS** is **unsigned long**. (These are defined in *vbus.h*)

Indented function names are intended only for system use.

For system administrators and maintainers, the *GRAMPS* file names are given below. The source is often very useful in understanding the behavior of the functions. The names of the files derive from the VMEbus implementation. There used to be two separate versions, one for the VMEbus, and one for the Multibus (called mbus). Later, the two were merged, but the names were not changed.

<<< vbusa.c >>>

vbusa.c contains those functions of the *GRAMPS* system that open or close 'files'.

open(*filename*, *rwflags*) **STR** *filename*; Open a file. If file is busy, keep trying. Equivalent to **openn**(*filename*, *rwflags*, 0). The second argument is used only for multiple-reader files; in other cases it must be present, but its value is irrelevant. An ((**int**) 'W') as the second argument means write privilege is being requested; the *Unix System V* symbols *O_WRONLY* and *O_RDWR* are also recognized as write requests. Any other value gets read-only. The symbols *W* and *R* are defined in **vbus.h** to have the appropriate values. (The explicit cast to **int** is not required when using the Intermetrics cross-compiler, but this is compiler-dependent.) Note: "w" (the string) is NOT a valid write argument. The *Unix System V* flags O_RDONLY, O_WRONLY, and O_RDWR are recognized, and this call is compatible with *Unix*.

openn(*filename*, *rwflags*, *ntries*, *fd_arg*) **STR** *filename*; **uint** *ntries*; Try to open a file *ntries* times (if busy). If there's an error, return(-1) immediately. If file is still busy after *ntries* tries, print a message and return(FILEBUSY) (FILEBUSY == -2). *ntries* == 0 means keep trying until open. The second argument, *rwflags*, is the same as for **open**. *fd_arg* is used only if *filename* == 0. It should be the value returned by a previous **get_fd** or successful **open** call. **openn** is preferable to **open** in finished systems, where a program which blocks indefinitely is not acceptable. The condition in which a file is busy for an unexpectedly long time requires corrective action by the affected process. However, for systems in a development phase **open** *is generally quicker to code (since there is no FILEBUSY return path), and is Unix*-compatible for testing on a *Unix* host system.

openn((**ADDR**) 0, *rwflags*, *ntries*, *fd*) **uint** *ntries*; If the *filename* argument is 0, then a fourth argument is present (used with **openn_fd**).

openn_with_previous_user(*filename*, *rwflags*, *ntries*, *previous_user*) **STR** *filename*; Open a file, but wait until the *previous_user* flag is *previous_user*.

openn_with_otheruser(*filename*, *rwflags*, *ntries*) **STR** *filename*; Open a file, but wait until the *previous_user* flag is other than this user.

openn_fd(*fd*, *rwflags*, *ntries*) **uint** *ntries*; Open a file, given its file descriptor (index into files array). This should be much faster than **openn**.

open_synch(*filename*, *rwflags*) **STR** *filename*; Wait until system clock is a multiple of *synchr_incr* offset from *synchr_base*, then open. This provides for synchronous communication, if desired. The variables *synchr_incr* and *synchr_base* are **extern unsigned int** and **extern unsigned long**, respectively, and may be changed by the user.

openr(*filename*, DUMMY) **STR** *filename*; open or return(-2) if busy. Use **openn**(*filename*, R, 1).

openw(*filename*, *ntries*) **STR** *filename*; **uint** *ntries*; Open for writing (applies to "multiple-reader" [simultaneous access] files). Use **openn**(*filename*, W, *ntries*). (**openr** and **openw** have been retained for compatibility with existing programs.)

(for system use:)

openonce(*filename*, *rwflags*, *ntries*, *fd_arg*) **STR** *filename*; Open or return (return -2 (FILEBUSY) if busy). *ntries* is vestigial; it used to be used for multiple-reader files. *fd_arg* is used only if *filename* == 0.

_open(*p*) **struct filebuf** **p*; System open.

_openwait(*p*, *ntries*) **struct filebuf** **p*; **uint** *ntries*; Keep trying to _open.

_openw(*p*, *ntries*) **struct filebuf** **p*; **uint** *ntries*; Open for writing.

close(*fd*) Close the file with file descriptor *fd*.

(for system use:)

_released(*p*) **struct filebuf** **p*; Free a *multiple-reader* file of this user's read and write bits.

_close(*p*) **struct filebuf** **p*;

flagpeek(*filename*) **STR** *filename*; Read the flag of the specified file without changing it.

flagpeek_fd(*fd*) Same, given the file-descriptor. Faster.

lastuser(*fd*) Returns the *previous_user* obtained from the system file flags. If the file is currently open to you, returns the user previous to you. (May need **get_fd**, below.)

get_fd(*filename*) **STR** *filename*; Get the file-descriptor of the named file. In *GRAMPS*, the file-descriptor is permanently associated with a named file, so this is meaningful even for an unopened file.

The following **preprocessor macros** are provided (in **vbus.h**) in order to insure portability of operations which form or decompose flagwords. This decomposition depends on the order in which bytes are concatenated to form words, which is different on an *8086* processor than on a *680x0*, for example.

FLAGBYTE(*flagword*) selects the byte that contains the OPEN/CLOSE information.

PREVUSER(*flagword*) selects the byte that contains the previous user's ID.

OPENFLAG(*userid*) constructs a flag word that says 'OPEN to *userid*'.

CLOSFLAG(*userid*) constructs a flag word that says 'CLOSED to *userid*'.

(for system use:)

struct filebuf ***getfptr**(*filename*) **STR** *filename*; Given a filename, return a pointer into the 'files' array.

<<< vbusb.c >>>

This file contains the file initialization functions.

filinit() Reset flags of all files owned or currently accessed by this processor. This function MUST be called by everyone at startup (and is called automatically in **vmain**). Current version zeroes all files whose flags are reset by this process (other than parmfile). Zeroing the files should make for easier debugging. If any flag is initialized, the remaining 14 bytes of the system's flagbuf are zeroed also.

(for system use:)

multinit(*p*, *flag*) **struct filebuf** **p*; **uint** *flag*; Called by **filinit** to initialize a multiple-reader file.

initopen(*addr*, *flag*) **ADDR** *addr*; **uint** *flag*; Return 1 for successful open, 0 if not successful.

zerofile(*p*) **struct filebuf** **p*; Zero file pointed to by *p* and its flagbuf, unless it's *parmfile* or *alloc**n.

getparmf() Locate *parmfile* in common memory, and return a pointer to it if it's there. Wait for it if so instructed.

getallfiles() Get *parmfile* and generate a 'files' array. For now, just read it all in into temporary (automatic) area. Later can screw around looking into file directly.

cpyfile(*p*, *a*) **struct filebuf** **p*; **struct allfilebuf** **a*; Copy a file's info from the 'allfiles' array to a

amongf(*id*, *bits*) **uchar** *id*; **USRBITS** *bits*; Is the userid *id* among those listed in *bits*? *id* is a userid like SSVA (0x2A); *bits* is *OR*'d from userbits like SSVABIT (1 << SSVAN, or 0x0002).

<<< vbusc.c >>>

vbusc.c contains *GRAMPS* functions other than 'file' I/O, including **printf**, **read0**, **write1**.

(for system use:)

STR libdate() Get *GRAMPS* date-of-last-revision.

char datchar(*d*) **long** *d*; Used by libdate.

printf(*control,args*, ...) **STR** *control*; The usual C print-to-terminal function. However, printing is skipped if no terminal is present. Note that not only is printing skipped, but formatting is skipped as well, thus removing the rather large overhead of the formatting whenever the terminal-device is disconnected.

dprintf(*control*, *args*, ...) Same as **printf**, except no printing takes place if *sysdebug* == 0. All the *GRAMPS* system programs call **dprintf**, so that system messages may be entirely turned off.

setcurrpic(*n*) **uint** *n*; Set the currpic field of the **PROC** array (*q.v.* Appendix B) to *n*.

exit(*i*) stop, and print a stack trace if *i* is non-zero.

abort(*s*) **STR** *s*; Stop, and print 'ABORT' and the specified (null-terminated) message.

(for system use:)

write1(*dummy*, *buffer*, *count*) **ADDR** *buffer*; Write a message to the terminal.

read0(*dummy*, *buffer*, *count*) **ADDR** *buffer*; Read and echo a line from the terminal. Return number of chars INCLUDING NEWLINE. Interpret line editing characters char delete and line delete.

readnl(*dummy*, *buffer*, *count*) **ADDR** *buffer*; Read from the terminal to (and including) the next newline. The returned count includes the newline (as on *Unix*). This is the same function as **read0** and **readline**.

readline(*dummy*, *buffer*, *count*) **ADDR** *buffer*; For compatibility with earlier forms.

changeproc(*userid*) **uint** *userid*; Change process identity (to *userid*) but still on same processor.

(for system use:)

getusern(*userid*) Get the user bit number *usern* (0 - 15) corresponding to the given *userid* (SSVA, e.g.).

sleep(*n*) *n* is approx in milliseconds. The system administrator should set the count on the **for** loop such that sleep(1) is approx 1 ms.

ulong wait_until(*t*) **ulong** *t*; Wait until the specified time, then return current time. If we're past the specified time return immediately.

ulong wait_to_multiple(*t*, *dt*) **ulong** *t*; **uint** *dt*; Wait until time is a multiple of *dt* offset from *t*, then return current time.

<<< vbusd.c >>>

vbusd.c contains **read**, **write**, and their analogs.

autoread(*filename*, *buffer*, *count*) **STR** *filename*; **ADDR** *buffer*;

autowrite(*filename*, *buffer*, *count*) **STR** *filename*; **ADDR** *buffer*; These two functions perform their own open and close.

read(*fd*, *buffer*, *count*) **ADDR** *buffer*;

write(*fd*, *buffer*, *count*) **ADDR** *buffer*;

readran(*fd*, *buffer*, *count*, *offset*) **ADDR** *buffer*; **long** *offset*; Direct access read ('readrandom') - read data starting *offset* bytes from the beginning of the file.

writeran(*fd*, *buffer*, *count*, *offset*) **ADDR** *buffer*; **long** *offset*;

ADDR readpointer(*filename*, *wronguser*) (formerly called **readheader**) **STR** *filename*; **char** *wronguser*; Check that someone OTHER than *wronguser* has used *filename* then read and return pointer.

writepointer(*filename*, *wronguser*, *pointer*) (formerly called **writeheader**) **STR** *filename*; **char** *wronguser*; **ADDR** *pointer*; Check that user other than *wronguser* has used previous pointer. Write new pointer.

(for system use:)

isopen(*p*) **struct filebuf** **p*; Is this file open to you, according to the system flags?

<<< vbuse.c >>>

vmain() Always called at startup by **pmain**, to initialize the *GRAMPS* system.

<<< vbusf.c >>>

alloccm, **freecm**, and related programs. Note the distinct names: **alloc** and **free** are the C- library routines that get space on the user's own board for its internal use; **alloccm** and **freecm** get space in common memory, which is accessible to other users.

ADDR alloccm(*size*, *uservec*) **uint** *size*; **USRBITS** *uservec*; Get a block of common memory, of the requested *size* (bytes). *uservec* is the bitwise *OR* of all userbits who will get use of this block.

freecm(*fp*) **ADDR** *p*; Return a block of common memory to the free list. **freecm** removes your userbit from the uservec; only if you are the last user does the block actually get returned to the free list.

addusers(*fp*, *addvec*, *subvec*) **ADDR** *fp*; **USRBITS** *addvec*, *subvec*; Update the bit vector of a block of memory by adding the bits in *addvec*, and removing those in *subvec* (which may include the current user itself).

freelink(*fp*, *forward_offset*, *backward_offset*) **ADDR** *fp*; NOT YET SUPPORTED. Return a link of a doubly-linked list to the free list, first fixing the links. *fp* points to the block (list element), *forward_offset* is the offset from *fp* of the pointer to the next element in the linked list, *backward_offset* is the offset from *fp* of the pointer to the previous element.

(called by system:)

initalloccm() Must be called by vmain.

freeinit(*usern*) Search list belonging to *usern*; free any blocks that have your usermask.

get_allocb() Get parmfile and generate an *_allocbase* array. For now, just read it all in.

markallocb() Mark your own entry of the COMMON MEMORY copy of *_allocbase* as being initted.

openalloc(*usern*) If speed is needed, this can become its own streamlined version of open. For now, use existing open (create entries in files array).

closalloc(*fd*) Close alloclist for this user.

getallocb(*fulladdr*) **p_header** *fulladdr*; Return userbit number in whose allocation area *fulladdr* is located.

STR allistname(*usern*) Return filename of allocation list for *usern*.

<<< vbusg.c >>>

Debugging tools.

tools() A standalone program that may be entered directly from the monitor. Tools provides debugging tools for understanding the current state of the user's program. It takes care not to clobber the user's stack or register save area. The tools themselves are listed on and selected from a menu, *q.v.* Figure 4.

menu() Prints the menu in Figure 4, for **tools**.

dispflags() Display the system's current file flags.

displfl_afa(*afileb*) Display one flag using the *allfiles* array. Called by **dispflags**.

displfl_fa(*fileb*) Display one flag using the *files* array. Called by **dispflags**.

char *fstr(*str,num*) Set up a filename string for printing.

dispfiles() Display the user's 'files' array in human-readable form.

STR flagletter(*flag*) **char** *flag*; Set up flag for printing; return pointer to the string.

STR subrname(*addr*) **ADDR** *addr*; Return the subroutine name that the address *addr* lies in, as a null-terminated string.

traceall() Print stack trace of the currently running program. This version is called by **exit** and **abort**, and ends with a call to **stop**().

trace() Print stack trace of a stopped program. Uses saved_A6, in the register save area.

symtabserv() Menu-driven server subprogram to access global-symbol-table information, including symbol names, addresses, and values, and to change those values.

struct symbol *getaddr(*string*) **STR** *string*; Given a symbol *string*, return a pointer to its entry in the symbol table. Return NULL if the symbol is not found.

ADDR getsubaddr() Request subroutine name, and return its address.

symtabmenu() Print menu for the symbol table server.

symeq(*s, t*) **STR** *s, t*; Return TRUE if null-terminated strings *s* and *t* are equal, FALSE if not.

ADDR ishexval(*str*) **STR** *str*; If *str* is a hex number in ASCII, terminated by 0 or '%' or ' ', return the number in binary; otherwise return 0.

ishex(*c*) **char** *c*; Return TRUE if *c* is a valid hex character, FALSE if not.

printsymtab() Print the entire symbol table.

printsymbol(*sym*) **struct symbol *sym**; Print one entry in the symbol table.

displaysymbol(*addr, inbuf, end*) **ADDR** *addr, end*; **char** *inbuf[]*; For the symbol whose address is *addr*, print its contents according to the **printf** specification *%spec* (in *inbuf*). Ask if the contents are to be modified, and if so, modify them. (The argument *end* is used only with the specification *%a*).

STR index(*c, s) char c*; **char *s**; return a pointer to the first occurrence of *c* in string *s*; or NULL if none.

STR symbolstring(*name*) **STR** *name*; Copy string and add terminating 0; note that subsequent calls will overwrite the copy.

char htoi(*s*) **STR** *s*; Return value (<256) of an ASCII string *s* taken to be one or two hex characters.

currstack() Print contents of stack, from current stack pointer back.

monitor() Print 'Monitor' and exit DIRECTLY to monitor, do not save registers in save area.

debugflag() Set debug flags in **PROC** array and in program.

dispPROC() Display current entries in this user's **PROC** structure in common memory.

subr() Eliminate a subroutine by placing an **rts** (return from subroutine) right at the beginning; or restore it by restoring its initial op-code. Works on subroutines beginning with 'unlk a6'-- this includes all Intermetrics-compiled C programs, and most *GRAMPS* assembly programs. The subroutine will prompt for the name of the target subroutine, and the function (eliminate or restore).

dispmem(*addr, n*) **ADDR** *addr*; Display *n* bytes of memory.

char makeprint(*c*) **char** *c*; Mask out the parity bit of the character argument *c*. If *c* is a printing character, return it; if *c* is non-printing, return '.' instead.

readnltest(*fd, buf, count*) **ADDR** *buf*; Read to newline, unless first character is ctrl-E or ctrl-T.

<<< vbusasm.asm >>>

C-callable assembly-language subroutines run on VMEbus.

pmain() This is the starting point for all C programs. It is never called by a user program.

movblk(*sourceaddr, destaddr, bytecount*); **ADDR** *sourceaddr, destaddr*; **uint** *bytecount*; does a block move (source and destination buffers must not overlap).

movblkfast(*sourceaddr*, *destaddr*, *bytecount*); **ADDR** *sourceaddr*, *destaddr*; **uint** *bytecount*; **movblkfast** is similar to **movblk**, but addresses MUST be even, and *count* MUST be a multiple of 4. There is NO checking. Count must be less than 65535.

movbigfast(*sourceaddr*, *destaddr*, *bytecount*); **ADDR** *sourceaddr*, *destaddr*; **ulong** *bytecount*; **movbigfast** is like **movblkfast**, except *count* is a long (up to 4 M). As in **movblkfast**, addresses MUST be even, and *count* MUST be a multiple of 4. There is NO checking.

zeroblk(*destaddr*, *bytecount*); **ADDR** *destaddr*; **uint** *bytecount*; **zeroblk** zeroes a block of memory.

lockexch(*address*, *number*, *type* [, *prevflag*]); **ADDR** *address*; **uint** *number*, *prevflag*; **lockexch** performs a locked exchange with a specified location: (Returns the number originally in specified loc) *type*=1 for byte, *type*=2 for word, *type*=4 for 'initexchange' (called to open a file during initialization) puts in userid IF flag was previously not open (note difference from **openexch**), and restores flag if 'INUSE'. *type*=5 for 'close-exchange' (called during closing of file). This function was very useful on the *8086XS* processor, which had locked (uninterruptible) exchange capability. On the *680x0*, it is easily replaced by **peeks** and **pokes**.

openexch(*address*); **ADDR** *address*; This is the subroutine that actually performs the **open**, on the flag at *address*. If the flag is already OPEN, return(FILEBUSY). Otherwise **open** it using a TAS (test-and-set), and put userid next to flag. In either case, update user's files.previous_user.

peek(*address*); **ADDR** *address*; returns(longword); Fetches the longword at location *address*. *address* must be even (except *68020*).

peekb(*address*); **ADDR** *address*; returns(byte); Fetches the byte at location *address*. Note that **peek** cannot be slipped in in place of **peekb**, since the desired byte is returned at the other end of the longword return register.

pokel(*address*, *longword*); **ADDR** *address*; **long** *longword*; Stores a longword at location *address*. *address* must be even (except *68020*).

pokew(*address*, *word*); **ADDR** *address*; **uint** *word*; Stores a word at location *address*. *address* must be even (except *68020*).

pokeb(*address*, *byte*); **ADDR** *address*; **char** *byte*; Stores a byte at location *address*.

regpeek(*register*); **int** *register*; returns(long); Returns contents of register (0-7 gets d0-d7; 8-15 gets a0-a7).

poke(*address*, *longword*); **ADDR** *address*; **long** *longword*; Same as **pokel**, for compatibility with existing programs.

systime() returns(longword). Returns system time.

imalive() returns system time(long). Sets user time to systime to show SYS that this process is still alive.

stop() **stop** is the simplest way to exit the program when something goes wrong, without doing a return to noplace.

getchar() returns(char). I/O primitive for reading from the terminal. Actually returns an int with high-order byte 0. (The earlier name **getcha** is also supported.) Control-C will cause the process to stop (by calling **stop**) as soon as it is read in, i.e. at the next **getchar**, **putchar**, or **ifchar**).

putchar(*char*); I/O primitive for writing to the terminal. (The earlier name **putcha** is also supported.) Control-S and control-Q may be used during output to turn transmission off and back on, respectively. (The program will block - pause - during this time.) Control-C will cause the process to stop (by calling **stop**) as soon as it is read in, i.e. at the next **getchar**, **putchar**, or **ifchar**).

getch() returns(char). **getch** is like **getchar**, but does not echo the character to the terminal.

ifchar() returns(char). Returns 0 if no character is waiting to be read from the terminal. Otherwise, returns the character but doesn't 'use it up': the next call to **getchar** will return the same character. Normally used to see whether a character is waiting, if you don't want to block (hang); **getchar** will block if there is no character (as in *Unix*).

testcrt() **testcrt** is a subroutine to test if a device (presumably a CRT) is hooked up to the UART port. Returns nonzero if a device is there, 0 if not.

checkstack() Checks the current stack pointer to see if the stack is infringing on program space. If less than 4k bytes remain, the program aborts. This number (4k) may be changed by changing the value of the symbol STACK_RESERVE in **vbusasm.asm**.

For SYS process only (and not currently implemented):

incrtimer() Interrupt service routine to increment system clock.

initintr() Called at startup to initialize interrupt chip (and sysclock for SUP).

resetm(*n*) Reset the board in slot *n* (*n* = 0, ..., 15).

initscope() Initializes the parallel output ports for use by **onscope** and **offscope**.

onscope(*arg*) Sets a specified bit of the output ports.

offscope(*arg*) Resets a specified bit. These three routines are used to toggle output bits at specified points in a program, so the program's progress can be monitored on an oscilloscope.

<<< vbussup.c >>>

This is kept as a separate source file and linked only with SUP. vbussup.c contains *GRAMPS* functions that only SUP uses. (including restart). It is not currently implemented.

resetall() Reset and restart all boards (operator can issue G xxxx).

restart(*id*) Reset and restart board with userid *id* (userid = 'ALL' works recursively).

resetonly(*id*)

(for system use:)

getslot(*id*, *PROCad*) **ADDR** ***PROCad***; Return slot number and **PROC** structure base address of the given *id*.

<<< scheduler.c >>>

This file and the next contain the primitives for running multiple tasks on a single processor. Most of these functions can only be called by the scheduler or by the system itself.

(only for use in **scheduler**:)

scheduler() - as the name implies, this function acts as the **main** program and calls the various tasks according to the algorithm programmed by the system administrator. It runs in privileged mode. It is called at the end of **vmain**, and should never return.

spawn(*process_start, n*) **ADDR** *process_start*; - Attach *context*[*n*] to the process which starts at *process_start*, and start it, returning at the end of that process's **vmain** (*i.e.* go through user's initialization).

<<< sched.asm >>>

relinquish() - issue a *trap 1* to voluntarily return control to the **scheduler**. This is the only user-callable multi-tasking function.

(only for use in **scheduler**:)

schedinit() - initialize the interrupt vectors for the scheduler clock and *trap 1*, etc.

activate(*nextcontext, time*) **struct context** ***nextcontext;*** - activate the task described in the indicated *context* slot. If *time* is nonzero, set scheduler clock to interrupt task after *time* msec.

start_scheduler_clock(*time*) - set scheduler clock to interrupt in *time* msec.

(for system use:)

schedisr - interrupt service routine for *trap 1* command. Save context of current task and restore context of **scheduler**. Return to **scheduler**.

sched_clock_isr - interrupt service routine for scheduler clock. Save context of current task and restore context of **scheduler**. Return to **scheduler**.

Appendix D - Local Environment - NBS

Currently, the development environment at NBS consists of two Dual Systems computers (*sight* and *vision*), connected to a VME-based system using a serial download link. (The speed of downloading will be greatly increased by using a parallel interface between the Duals and the VME bus, which is expected shortly). Programs are developed in *C*, compiled on the Duals using the Intermetrics *C* cross-compiler, and linked with the *GRAMPS* library. Only the *C* language and *68000* assembly language are currently supported. Substantial effort would be required to support other environments and languages.

The choice of cross-compiler fixes certain parameters, such as the lengths of variables. Thus, for the Intermetrics compiler, a **char** in the target system is 8 bits long, and both signed and unsigned versions may be used. **short** integers are also 8 bits long. A regular **int** is 16 bits long, and may also be signed or unsigned. **long** integers are 32 bits in length, as are all pointers. The extra codes used by the *68020* and the 68881 coprocessor are not yet supported by the Intermetrics compiler, but should be in the near future.

The library containing *GRAMPS* is located in the directory */a/vlib* on both *sight* and *vision*. The source programs are in */a/vlib/vbus*, and are called **vbusa.c**, through **vbusg.c**, with assembly language subroutines in **vbusasm.asm**. The current **#include** files are in /a/vlib. Some of these are discussed further in the *Administrators Guide*.

To compile a *C* program using the Intermetrics cross-compiler, type

ccv *file1.c file2.c ...*

Assembly-language programs work much the same, except using **asmv** instead of **ccv**. In this case also the extension **.asm** is optional:

asmv *file3*

ccv and **asmv** both create an object file with suffix **.ol**. The object files then have to be linked together along with the *GRAMPS* library using **llf**:

llf *file1 file2 file3 ...*

llf stands for link, locate, and format. It leaves its output ready for downloading in a **.dn** file, in *GRAMPS* **.dn** format. **ccv, asmv,** and **llf** are in the directory */a/x*, which should be included in the user's PATH environment variable (in his *.login* or *.profile* file, as applicable).

Appendix E - Local Environment - Philips

Currently, the development environment at Philips consists of a Sun workstation (*jay*), connected to a VME-based system using a serial downloader. Programs are developed in *C*, compiled using the Intermetrics *C* cross-compiler, and linked with the *GRAMPS* library. It is expected that the download capability will be improved, either by using a parallel interface to the VME bus, or by direct memory transfer.

The choice of cross-compiler again fixes certain parameters; see Appendix D above.

The source code and library containing *GRAMPS* is located in the directory */usr/local/interc/glib* on *jay*. The source programs are called **vbusa.c**, through **vbusg.c**, with assembly support in **vbusasm.asm**, **pmain.68k**, and **mmu.68k**. Those with strong constitutions might want to examine some of these files. The corresponding **#include** files are in */usr/local/interc/include*. Some of these are discussed in the *Administrators Guide*.

To compile a *C* program, or to assemble an assembly-language program, using the Intermetrics tools type

ic68 -G -o *name file1.c file2.c ...*

where the **-G** flag includes the *GRAMPS* library, the **-o** flag renames the output to *name*, and *file1.c*, *file2.c*, ... are file names. Assembly programs can also appear in the command line, and will be assembled and linked. A manual page will be forthcoming on this command. **ic68** is in the directory */usr/local/interc/bin*, which should be included in the user's PATH environment variable (in his *.login* or *.profile* file, as appropriate).

The following programs are specific to the Philips implementation of *GRAMPS*. Similar programs may be necessary for each system, but will in general be hardware-specific.

<<< mmu.68k >>>

This assembly-language program sets up the memory mapping in the memory-management unit. It also currently initializes some of the "magic" numbers and locations.

<<< mkversion and version.c >>>

In order to establish the date of last update of the *GRAMPS* system, a system date function is included in the library, and printed out on startup of each process. **mkversion** creates a *C* program in the file **version.c**, which is compiled and added to the library automatically with each use of **candl** or **aandl**.

REFERENCES

[1] Mansbach, Peter, *Overview of the 'GRAMPS' Multiprocessor Operating System, (to be publ.)*, 1988.

[2] Mansbach, Peter, and Shneier, Michael, *The GRAMPS Operating System: Administrator's Guide*, National Bureau of Standards, Gaithersburg, MD 20899 *(to be publ.)*, 1988.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 88-3778	2. Performing Organ. Report No.	3. Publication Date SEPTEMBER 1988

4. TITLE AND SUBTITLE

The GRAMPS Operating System: User's Guide

5. AUTHOR(S)

Peter Mansbach and Michael Shneier

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Manufacturing Engineering
Robot Systems Division
Gaithersburg, MD 20899

10. SUPPLEMENTARY NOTES

[XX] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This guide describes the GRAMPS real-time multiprocessor operating system from an applications viewpoint. It presents the information needed to use GRAMPS in implementing distributed processing applications. Additional information needed by an administrator to set up and maintain a specific application appears in the Administrator's Guide.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

asynchronous communication; communications protocol; functionally-divided processes; GRAMPS; multi-processor; multiprocessing; multiprocessor; operating systems; real-time; robot vision; vision

13. AVAILABILITY

[X] Unlimited

[ ] For Official Distribution. Do Not Release to NTIS

[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

[XX] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

42

15. Price

$11.95

USCOMM-DC 6043-P80